SEF - Week 3 Class diagrams

Introduction

As well as modelling the behaviour of a system - what features it provides and what benefits it gives to its users - we also need to model the data involved in the system.

This modelling is often split in two - the domain model and the design model.
The domain model describes the problem domain as the client understands it. For an accountant it includes invoices and receipts and customers and taxes. For a hospital it includes patients and consultations and prescriptions and operations.

The design model is created by the developers to model the solution, and typically subsumes the domain model while adding in features needed to implement it such as databases and networks and user interfaces.

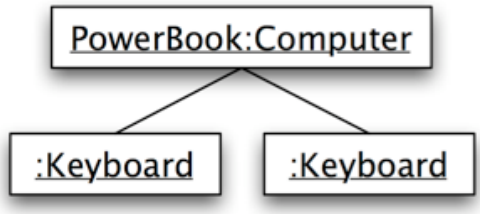
The data we want to model often falls naturally into categories - this classification leads to ***classes***. Each class describes the common characteristics of the things we want to model.
The Patient class in a hospital system would include their name and address and insurance details. An invoice class would include an amount, invoice number and date.

If we examine the system we are modelling - which might be a currently running company, or a computer system which might be built - we would find ***instances*** - each an occurrence of a class.
Each instance or ***object*** models a single entity. A hospital that has 20 patients would have 20 patient objects, each object detailing the patient's details (known as ***attributes)*** such as name, date of birth, sex etc..

As well as modelling entities/objects by describing them and their attributes in a class definition, we also need to model relationships between these objects. For example a doctor provides a treatment for a patient - we can model the association between doctor and treatment ("doctor-provides-treatment") and treatment and patient ("patient-receives-treatment").

Object diagrams

UML provides the class diagram and the object diagram for modelling the data in a system. When modelling a real system each entity/thing will have exactly one counterpart in the object model.



Here is an example of a laptop computer with a built in keyboard, and an attached keyboard. The object diagram has a box for each major item that we want to model - two keyboards and the computer (note that we choose not to model the individual keys, nor the trackpad or display etc. - we obviously have to stop somewhere when modelling!).

Lines join the "Powerbook: Computer" object to the two ":Keyboard" objects. These show that the objects are linked in some way (for this example it is showing that they are physically connected) - in fact lines between objects in an object diagram are known as links.

As well as showing that items are physically connected, links can also show other types of connections - such as family relationships (brother, cousin, mother), location relationships (adjacent to, above, below) etc.

The naming of the objects is specified by UML - "Powerbook : Computer" says that the object (indicated by the underline and the ":") has been placed into the classification "Computer" (class Computer), and been given the unique name "Powerbook".

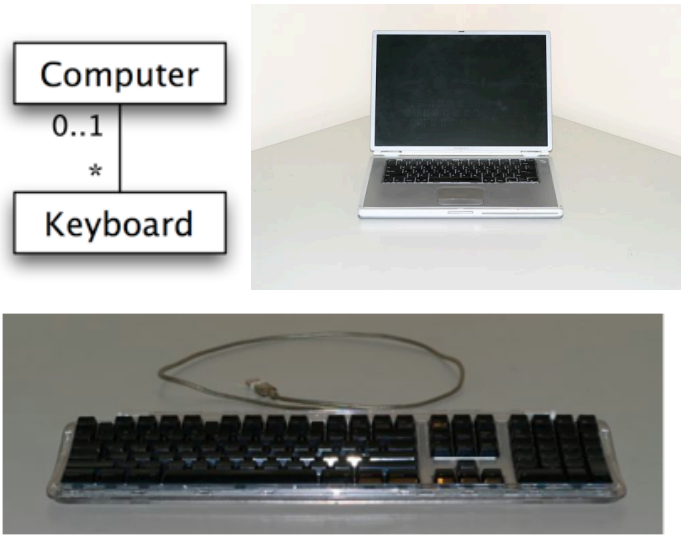
The two keyboards are from class "Keyboard" and are anonymous objects - unnamed but still unique (just as the keyboards in the photo are unique).

We create object diagrams to show the ***valid*** configurations we expect the system to take when running. Examples of invalid configurations for a hospital would be a patient who was discharged before they were ever admitted, or were admitted after they had died. Similarly an invoice from a company should always never contain multiple customers.

Class diagrams

The previous object diagram represented one configuration of objects and their relationships. Obviously we can have a huge number of configurations - so many in fact that attempting to draw all possible valid configurations is impossible.
Instead we attempt to generalise - to draw a diagram that shows the valid objects and associations between them, as well as any other restrictions or constraints we may want to place on them.

A class diagram is an abstraction of all the possible object diagrams that we are interested in modelling.



This diagram shows class Computer is connected to class Keyboard via an ***association***.
The class is a generalisation of all the objects we possibly want to model.

How to select classes can be tricky - sometimes the borders between different possible classes is not as clear cut as you would hope. For computers do we want to consider desktop computers, portables and servers as being the same? Are there some characteristics relevant to the problem being modelled (attributes or the way that they behave) that would require them to be classified differently?
An increasing number of devices now have processors embedded in them - games consoles, DVD players, mobile phones - should they be characterised as computers in our system?
In the end the decision about how we split the world we want to model into different categories (classes) will be driven by the pragmatic concerns of our client's needs, future plans and our experience in creating versatile abstractions.

Constraints

It is interesting to note that a large part of specifying a system is in devising restrictions. The name of a person is restricted to characters (not numbers or dates). The age of a person is restricted to 0 to ~110. Specifying a ***type*** for an attribute is placing restrictions on the of values it can have.

Breakout box (list)
Common types for attributes are numeric (real and integer), character, string (collection of characters), date, time, boolean (true/false)
****

The objects in a computer could be allocated any value we like from the extremely large universe of possible values - our job is to constrain and restrict these values to those that model the system we want.

We are constantly narrowing down the range of values objects can have, and the associations they can maintain - removing the freedom to be anything they want until we eventually get to just those allowed by the system we are modelling.

When thinking about associations we need to consider how many objects can be involved in each association. For example how many beds can a patient be in at one time? How many doctors can be involved in a patients treatment? The most common answers for these questions are 1, 0 or 1, 0 upto many and 1 upto many, but other answers are possible.
For example the USB standard for connecting devices to a computer specifies a maximum of 127 devices, while Firewire supports 63.

Specifying association constraints

A constraint on an association is specified by annotating each end of the association line. The class diagram above (Computer - Keyboard) shows 0..1 and *. "0..1" shows that a keyboard can be associated with 0 or 1 computers. "*" indicates 0 upto many ("many" really means no restriction).
If the diagram does not have an explicit constraint then the implicit constraint is 1 i.e. a mandatory association.

Example association constraints

A person can have 2 biological parents (2).
A student can be enrolled in 1 to 6 subjects (1..6)
A computer can be connected to up to 127 USB devices (0..127)
An invoice has at least one billable item, but may have many (1..*)

Roles

Classes at each end of an association perform a particular ***role***. For example the roles on the association between the classes Person and Property could be "owner" and "is owned", or perhaps "occupant" and "rental property".

More on associations

We now look at ways in which associations can be used in class diagrams.

Self association

A class can reference itself, for example the association "parent-of" is between Person and Person. Properties could have an "adjacent-to" association, computers can have a "connected-to" association and so on.

It may be tempting in the parent-of association to place a constraint of "2" (everyone has two (biological) parents). However this is a constraint that must be maintained by any system we develop. For every Person you record in a computer system can you identify who their parents are (and their parents, and so on ad-infinitum)? Not enforcing this constraint means a broken program - the modelling and the behaviour of the system don't match up.
In this case the association has to be "relaxed" to be "known-parents-of" which involves a constraint of 0..2, and is therefore implementable.

Generalisation/inheritance

Often a number of objects we want to model are similar, but have different attributes, or different behaviours. We would like to capture this commonality (why would we want to throw away this information?) but not by ignoring the differences.

Inheritance (or generalisation - the same concept but viewed from the opposite viewpoint) is another type of relationship that can be modelled in class diagrams (associations is the other type of relationship that has been presented).

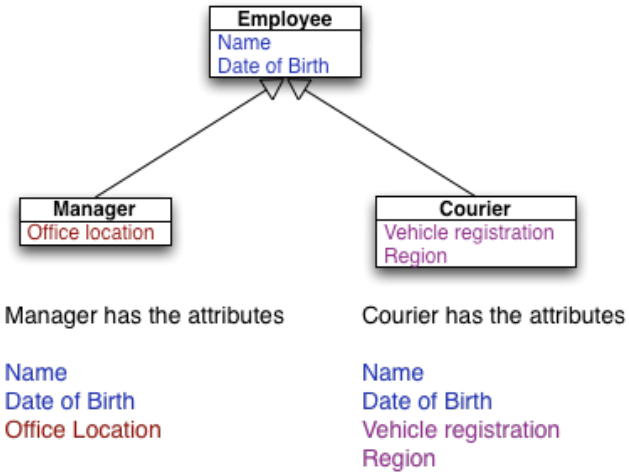
It allows you to link classes which have common features by describing a parent class that serves as a focus point for those features, while child classes add new features (such as attributes or behaviours), or override existing behaviours to make them different to the parent class.

Parent and child classes are also known as super and sub classes.

A diagram of inheritance usually shows a branching layout - at each level of inheritance there are more children than parents, much like an upside down tree.

Java (an object oriented programming language - one that has classes and inheritance) has a hierarchy of classes. At the very top of the inheritance hierarchy (or tree) is the class Object. Below that there is a multitude of classes. The windowing system (for user interfaces) relies heavily on this approach.
The JPanel class (used for alerts and pop up boxes) is a subclass JComponent, which is a subclass of Container, which is a subclass of Component, which is finally a subclass of Object. At each step down the inheritance tree there are more and more features and behaviours added. Despite their different appearance, buttons that allow you to click "Ok" or "Cancel" share a JPanel's inheritance path.

Another, more commonplace example, is of the class Employee, with attributes of Name, "Date of birth" etc. Subclasses could be Manager who adds the attribute "Office location" and "Courier" who adds the attribute "Vehicle registration" for the truck he drives, and Region - which area he delivers into.



Inheritance is shown as an open triangle pointing at the parent class with a line leading to the child class(es).

Abstract classes

In the Employee example above there may be a more specific classification for *all* of the staff than just "Employee". For example there may be Secretary, Cleaner, Packer, Manager and Courier. If you have a situation like this where you don't expect to ever

have a "raw" Employee object, you can define it to be ***abstract***. Naming a class to be abstract (this is actually applying what UML calls a ***stereotype***) states that objects of this class should never be created.

Taxonomies in the natural world show this - all but the ***leaf*** nodes (the very extremities of the taxonomic tree) are considered abstract.
People are classified as Primates, which are classified as Mammals, which are classified as Chordates etc., but individual people would be labelled as "Person", not "Primate", or "Mammal".

Essentially an abstract class is an artificially created placeholder to express the commonality of several real classes, and doesn't relate to any real concept in the world we are modelling.

Attributed associations

Students enrolled at university take many subjects and obtain a result for each subject. Where should that result be stored?
If it is stored within the student, then there would be no way to determine which subject the result was for. If it was stored in the subject, there would be no way to determine which student it was for.
The result is actually an attribute of the pair Student/Subject, not of either one individually. The attribute "result" needs to be stored with the thing common between them - the association.

UML allows you to draw an extra class - the association class - that is attached to the association via a dashed line. Objects of this class can only exist when a link (an instance of the association) exists between the main objects exist.

Composition and aggregation

The concepts of composition and aggregation have little reason to recommend them, apart from some very specialised corner cases related to transitivity (see Priestley for a further explanation). They are meant to represent a "part-of" relationship between classes - such as an engine is a part-of a car, or an order-line is part-of an order.

However all of the claimed semantics for composition and aggregation are met by simple constraints such as "1"!

The topic is stated here as many people feel strongly that these concepts are important, and you do need to be aware of them.